A Resource Management Framework for Central Servers

A Resource Management Framework for Central Servers

David G. Sullivan, Margo I. Seltzer
Technical Report TR-13-99
Division of Engineering and Applied Sciences
Harvard University, Cambridge, MA 02138
{sullivan,margo}@eecs.harvard.edu

Abstract

Proportional-share resource management is becoming increasingly important in today's computing environments. In particular, the growing use of the computational resources of central service providers argues for a proportional-share approach that allows clients to obtain resource shares that reflect their relative importance. In such environments, clients must be isolated from one another to prevent the activities of one client from impinging on the resource rights of others. However, such isolation limits the flexibility with which resource allocations can be modified to reflect the actual needs of clients. We present extensions to the lottery-scheduling resource-management framework that increase its flexibility while preserving its ability to provide secure isolation. To demonstrate how this extended framework safely overcomes the limits imposed by existing proportional-share schemes, we have implemented a prototype system that uses the framework to manage CPU time, physical memory, and disk bandwidth. We present the results of experiments that evaluate the prototype, and we show that our framework enables clients of central servers to achieve significant improvements in performance.

1 Introduction

In managing computational resources, an operating system must balance a variety of goals, including maximizing resource utilization, minimizing latency, and providing fairness. The relative importance of these goals for a particular system depends on the nature of the system and the ways in which it is used. For supercomputers running compute-intensive applications, for example, the primary goal may be to maximize throughput, while for personal computers used to enhance a single user's productivity, the chief goal may be to maximize responsiveness.

In today's computing environments, users increasingly compete for the resources of server systems, whether to access central databases or to view content on virtually-hosted Web sites. On such systems, fairness becomes a critical resource-management goal. Proportional-share mechanisms allow this goal to be met by providing clients (users, applications, threads, etc.) with specified resource shares. For example, customers who pay Internet service providers to virtually host their Web site can be guaranteed a share of the hosting machine that is commensurate with the price they pay. Service providers who can make such guarantees can offer larger resource shares to clients willing to pay a premium for better quality of service.

Although its full promise is yet to be realized, thin-client computing is another domain in which proportional-share resource management is desirable. Administrators of such systems are often forced to host one application per server to provide predictable levels of service [Sun98]. Proportional-share techniques enable the consolidation of multiple applications onto a single server by giving each of them a dedicated share of the machine.

Systems that support proportional-share resource management must *isolate* resource principals from each other, so that a given client's resource rights are protected from the activities of other clients. To provide such isolation, a system must necessarily impose limits on the flexibility with which clients' resource allocations can be modified. Such limits work well when the resource needs of clients are well-known and unchanging, since a system administrator can assign the appropriate resource shares and leave the system to run. Unfortunately, these conditions frequently do not hold. Even if a client's resource needs are adequately understood, they will typically change over time. For example, as a Web site's working set of frequently accessed documents expands, the site may require an increasing share of the server's disk bandwidth in order to offer reasonable responsiveness. Moreover, it would be preferable if system administrators could be freed from the need to make detailed characterizations of clients' resource needs. Ideally, the clients themselves should be able to modify their own resource rights in response to their needs and the current state of the system.

In this paper, we present extensions to the lottery-scheduling resource-management framework [Wal94, Wal95, Wal96] that enable it to securely support proportional sharing of multiple resources and that allow clients to safely overcome the limits on flexible allocation that lottery scheduling and other resource-management frameworks impose for the sake of secure isolation. Our extended framework supports both absolute reservations and relative resource shares, and it provides a system of access controls to protect the isolation properties that the framework provides. In addition, our framework allows clients to take advantage of their differing needs and to coordinate their use of the system's resources by bartering over resource rights with each other. Clients can thereby modify their own resource rights without compromising the rights of other clients. Our extended framework

thereby provides isolation with increased flexibility, enabling a client-centered approach to resource management on server systems.

We have developed a prototype implementation of this extended framework in the VINO operating system [Sel96], and have used it, in conjunction with several proportional-share mechanisms, to manage CPU time, physical memory, and disk bandwidth. Our experiments demonstrate that the extended lottery-scheduling framework enables clients of central servers to achieve improved performance under realistic usage scenarios.

This work makes several contributions. First, we extend the lottery-scheduling framework to securely support management of multiple resources, providing both soft and hard resource shares. To our knowledge, our prototype is the first implementation of a proportional-share resource-management system to support both absolute and relative resource shares for multiple resources. Second, we point out an important tension between the conflicting goals of secure isolation and flexible resource allocation, and we present mechanisms for providing greater flexibility while preserving secure isolation. Third, we illustrate the value of a system that supports dynamic adjustments to the resource allocations that applications receive.

In the next section, we situate our work by describing the original lottery-scheduling framework and illustrating how it imposes both upper and lower limits on the resource allocations that clients can receive. We also discuss other proportional-share resource-management schemes, comparing them to lottery scheduling and showing that they impose similar limits on flexible allocation for the sake of secure isolation. In Sections 3 and 4, we describe our extensions to the lottery-scheduling framework and explain how we overcome both sets of limits while maintaining secure isolation. In Section 5, we describe our prototype implementation of the extended framework, including the scheduling mechanisms that we have chosen to employ. Section 6 presents experiments designed to evaluate the prototype and to test one of our mechanisms for increased flexibility. Finally, we summarize our conclusions and present ideas for future work.

2 Background

2.1 The Lottery-Scheduling Framework

The resource-management framework developed for lottery scheduling [Wal94, Wal95, Wal96] is based on two key abstractions, *tickets* and *currencies*. Tickets are used to encapsulate resource rights. Clients receive resource rights that are proportional to the number of tickets that they hold for that resource; changing a client's ticket holdings automatically leads to a change in its resource rights.

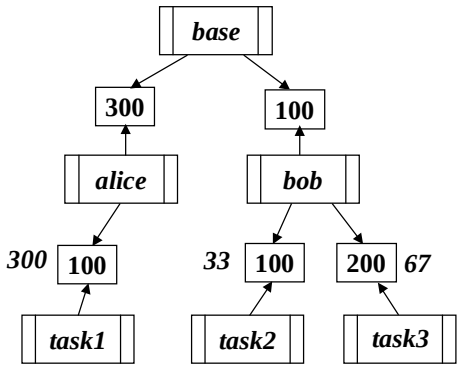

Figure 1. A sample resource hierarchy in which currencies provide isolation between the tasks of different users. The base values of the tickets held by the tasks are shown in italics.

Tickets are issued by currencies, which allow clients to be grouped together and isolated from other clients. Clients funded by a currency share the resource rights allotted to that currency; currencies thus enable hierarchical resource management. Each currency effectively maintains its own exchange rate with a central base currency, and tickets from different currencies can be compared by determining their value with respect to the base currency (their *base value*). The more tickets a currency issues, the lower their base values, and their total base value can never exceed the base value of the tickets used to back the currency itself. For instance, in Figure 1, the *bob* currency is funded by 100 of the 400 base-currency tickets, and it thus receives rights to one-quarter of the resource. These rights are divided up by the tasks funded by *bob;* for example, *task3* holds 200 of the 300 *bob* tickets, and it thus receives rights to two-thirds of *bob*'s quarter share, or one-sixth of the total resource rights. In other words, *task3*'s 200 *bob* tickets have a base value of approximately 67 (two thirds of 100). If *task2* or *task3* forks off additional processes, causing the *bob* currency to issue more tickets, the value of its tickets will decrease, since its resource rights will be divided up among a larger number of clients. However, the resource rights of processes funded by other currencies will not be affected.

While a lottery-scheduling resource hierarchy typically has a tree-shaped structure like the one shown in Figure 1, it can more generally take the form of any directed acyclic graph. The lottery-scheduling framework thus supports a greater variety of configurations than most other, recently proposed schemes for hierarchical resource-management (see Sect. 2.3). For example, on a system like the one depicted in Figure 1, in which each user's applications are funded by a currency specific to that user, two or more users could pool their resources to support a single application that all of them are using ([Ban99] also allow this).

Tickets and currencies can be used to specify "soft" resource shares—relative shares that vary with the number of active clients—as well as "hard" shares, or reservations, that represent a fixed percentage of the resource.Such fixed reservations are needed by many of today's resource-intensive applications, including real-time video and audio. To our knowledge, prior lottery-scheduling implementations [Wal94, A-D97, Pet99] have only supported soft shares. Our extended framework enables the use of hard shares as well, modifying and generalizing an approach outlined by Waldspurger and Weihl [Wal96] (see Sect. 3.3 for more details).

2.2 Limits Imposed by Currencies

Currencies, like all mechanisms for providing isolation, necessarily impose limits on the flexibility with which clients' resource allocations can be modified. In this section, we demonstrate that currencies enforce both upper *and* lower limits on resource allocations.

2.2.1 Upper Limits. The resource rights of a client funded by a subcurrency (i.e., a currency other than the root currency) can usually be increased by giving the client additional tickets from that subcurrency.[1] For example, in Figure 1, *task2*'s resource share could be boosted by giving it 200 *bob* tickets rather than 100. However, this doubling of the subcurrency tickets that *task2* holds does not double its resource rights; rather, *task2* goes from having one-third of the *bob* currency's overall resource share (a base value of 33) to having one-half of *bob*'s overall share (a base value of 50). This smaller increase is a result of the fact that issuing additional *bob* tickets decreases their value. No matter how many subcurrency tickets a client receives, the resource rights imparted by those tickets cannot exceed the overall rights given to the subcurrency. This upper limit is essential to providing isolation. Without it, the resource rights of clients funded by other currencies could be adversely affected.

Despite the need for these upper limits, there may be situations in which the resource rights protected by currencies do not meet the clients' actual resource needs. In such cases, it may be possible to safely overcome these limits in order to better meet clients' needs. Section 4 describes the mechanism we have developed for this purpose.

2.2.2 Lower Limits. Currencies can also impose lower limits on the resource rights of clients holding their tickets. These limits materialize when only one of the clients funded by a subcurrency is in active contention for a given resource.[2] In such circumstances, the client receives *all* of that subcurrency's resource rights, no matter how few subcurrency tickets have been used to fund it.

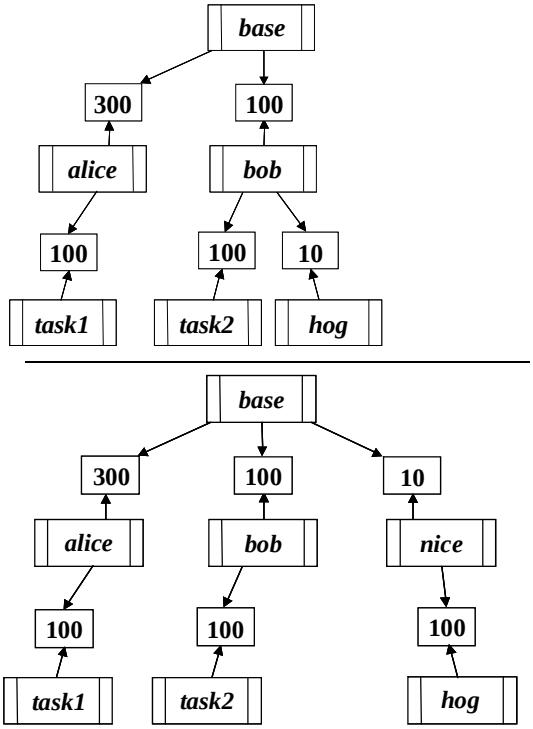

Figure 2. The user *bob* attempts to lower the priority of *hog*, a CPU-intensive process, by giving it only 10 tickets (*top*). However, if *task2* becomes idle, *hog* will still receive all of Bob's resource rights. One way to solve this problem is to fund *hog* directly from the base currency (*bottom*).

As a result, subcurrencies make it difficult for the lottery-scheduling framework to support the semantics of the nice (1) utility found on conventional UNIX systems. For example, a user running a lengthy, CPU-intensive job may reduce its CPU funding as a favor to other users. But if the other tasks funded by the same subcurrency are all idle, the CPU-intensive job will still get the subcurrency's full CPU share (Fig. 2, *top*). The user would presumably be allowed to decrease the number of tickets backing the subcurrency itself, but then the other clients funded by that subcurrency would also be affected when they reentered the competition for CPU time.

While upper limits are necessary for providing isolation, lower limits are an undesirable side-effect of isolation. These limits could be overcome by funding CPU-intensive applications directly from the base currency (Fig. 2, *bottom*). However, for reasons of security,

1. This is not always the case. If a client is the sole recipient of a subcurrency's tickets, giving it more tickets from the subcurrency does not affect its resource rights.

2. When a client temporarily leaves the competition for a given resource—e.g., when a thread blocks—its tickets become *deactivated* and the resource rights of other clients funded by the same currency are increased.

unprivileged users would presumably not be allowed to issue base-currency tickets. In Section 4.4, we describe one solution to this problem.

Before detailing the extensions we have developed to address these limits and increase the flexibility of the lottery-scheduling framework, we first briefly review other means for implementing proportional-share resource management, comparing them to lottery scheduling and noting that they impose similar limits on flexible allocation.

2.3 Other Resource-Management Frameworks

In addition to lottery scheduling, other frameworks can be used to provide proportional-share management of multiple resources. In particular, Rialto's *activities* [Jon97], Eclipse's *reservation domains* [Bru98, Bru99a], the *Software Performance Units (SPUs)* of Verghese et al. [Ver98], and the *resource containers* of Banga et al. [Ban99] function similarly to currencies in their ability to isolate single clients or groups of clients from each other.

Reservation domains and resource containers also share the lottery-scheduling framework's ability to support hierarchical resource management. However, the hierarchies supported by reservation domains are limited to a tree-shaped structure, in which the resource shares of non-leaf domains or containers are divided among their children. As discussed in the previous section, lottery-scheduling's currency hierarchies are more flexible, since clients can be funded by more than one currency and thus share the resource rights of multiple currencies. One important benefit of this more flexible scheme is that priority inversion can be easily handled using ticket transfers: when a client blocks to wait for a shared kernel resource, its tickets are temporarily transferred to the client currently holding the resource [Wal94]. Resource containers allow clients to be multiplexed over several containers and to receive their combined allocations, so they could potentially be used to support the equivalent of ticket transfers.

Moreover, most of these alternative frameworks only support fixed shares; clients without reservations either share the remaining CPU capacity equally (as in Rialto and Eclipse) or are scheduled according to a traditional time-sharing scheduling discipline. Lottery scheduling, on the other hand, can support both fixed and relative shares. Resource containers were not designed for one fixed use. In their prototype implementation, containers were used in conjunction with both fixed-share CPU guarantees and time-sharing, but they could potentially be used to support relative proportional-share guarantees.

The alternative frameworks do provide advantages over a traditional lottery-scheduling framework. In particular, activities and resource containers offer finer-grain resource management, addressing applications such as Web servers in which a single thread is associated with more than one activity. In addition, resource containers account for kernel-mode processing done on behalf of an activity. Our current implementation does not support these features, but we plan to extend our lottery-scheduling framework to support them.

Regardless of the framework used to provide proportional-share resource management, the need to isolate clients from each other necessarily involves imposing limits on resource allocations of the types described in the previous section. Clients restricted to a particular activity, reservation domain, SPU, or resource container cannot obtain more than their group's overall resource rights. If only one client sharing a reservation is actively competing for the reserved resource, it will receive the entire reserved share, even if it would be preferable for it to receive less than that amount. Methods for safely overcoming these limits like the ones discussed in Section 4 would be needed to allow these frameworks to provide greater flexibility while preserving secure isolation.

3 Secure Multi-Resource Lottery Scheduling

Before addressing the limits imposed by currencies, we first needed to extend the lottery-scheduling framework to securely support proportional sharing of multiple resources. The following sections describe these initial extensions, including the access controls needed to protect the isolation properties of currencies.

3.1 Resource-Specific Tickets

Although prior implementations of lottery scheduling have focused exclusively on single resources (primarily CPU scheduling), the original lottery-scheduling framework was designed to support multiple resources. Waldspurger [Wal95] considered two options for implementing a multi-resource system. In the first option, tickets are applicable to any resource, allowing clients to shift tickets from one resource to another as needed, while in the second, tickets are resource-specific. Waldspurger favored the former approach because of its greater flexibility and simplicity. However, allowing clients to devote tickets to resources as they see fit violates the insulation properties of currencies, since it can lead to changes in the total number of tickets applied toward a given resource, as the example in Figure 3 demonstrates.

We therefore chose to use resource-specific tickets. To avoid the overhead of maintaining a separate currency configuration for each resource, we extend currencies to encompass all of the resources being managed. Concretely, this means that most pieces of currency state, such as the list of issued tickets and the count of active tickets, are maintained as arrays indexed by resource type. Simi-

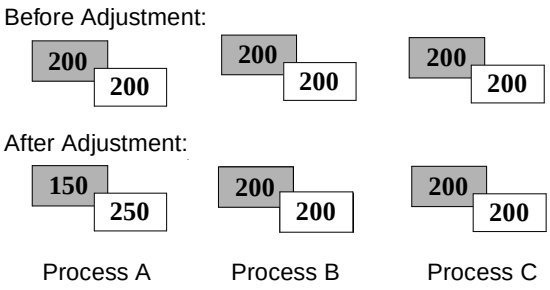

Figure 3: A Problem with Allowing Tickets to Compete for Any Resource. Process A shifts 50 of its tickets from one resource to another. Other processes now have a smaller percentage of the tickets devoted to the second ("white ticket") resource, and thus fewer rights to that resource.

larly, many currency-related operations take a parameter that specifies the resource type.

3.2 Currency Brokers

In order for the lottery-scheduling framework to be secure in a multi-user setting, a system of access controls are needed. We encapsulate these controls in a *broker* associated with each currency. A broker stores the owner and group of the user who created the currency, along with a UNIX-style mode specifying who may perform various operations on the currency. Before these operations are carried out, the broker verifies that the current thread belongs to a user with the requisite permissions.

Like UNIX file modes, currency modes include three sets of permissions, one for the currency's owner, one for the currency's group, and one for all others. In a given set of permissions, the *f* bit indicates whether a user is allowed to fund the currency; the *c* bit indicates whether a user can "change" the currency by removing some of its funding or destroying it entirely; and the *i* bit indicates if a user is allowed to issue or revoke the currency's own tickets. Table 1 provides more specifics about the permission checks that brokers perform. In most cases, superusers are allowed to override the ordinary permissions checks. If an attempt to fund a currency would lead to a cycle in the currency graph, the attempt is rejected.

3.3 Hard and Soft Resource Shares

The standard lottery-scheduling framework was primarily designed to support "soft" resource shares, i.e., shares whose relative values are constant but whose absolute values may change over time as clients enter or leave the competition for the resource. However, Waldspurger and Weihl pointed out that absolute, "hard" resource shares can be supported using the same framework by fixing the total number of issued tickets in the system [Wal96]. In particular, they proposed specifying absolute shares by issuing tickets from "hard" subcurrencies that maintain a fixed exchange rate with the base currency. When a hard currency issues additional tickets, some of the funding of other, "soft" currencies is transferred to the hard currency so that its exchange rate can be maintained.

Table 1. Permission checks performed by currency brokers

operation	permission check
create a currency	the caller must match both the user id and group id specified for the new currency
destroy a currency	the appropriate *c* (*change*) bit must be set in the currency's mode
fund currency A with tickets issued by currency B	the appropriate *f* (*fund*) bit must be set in A's mode **and** the appropriate *i* (*issue*) bit must be set in B's mode
take tickets issued by currency B away from currency A	the appropriate *c* (*change*) bit must be set in A's mode **or** the appropriate *i* (*issue*) bit must be set in B's mode.

In our extended framework, we take a slightly different approach based on the notion of hard and soft *tickets*, and we allow clients to obtain hard shares from subcurrencies as well. Under our approach, tickets issued by a currency are ordinarily soft tickets that specify relative shares of the currency's resource rights. However, when a currency issues a hard ticket to specify a fixed percentage of its resource rights, a separate soft-ticket subcurrency is created and used to fund the currency's soft tickets (Fig. 4). The number of hard tickets used to fund this soft-ticket subcurrency is adjusted as needed to ensure that the total number of the currency's hard tickets remains fixed.

Our approach has the advantage of requiring no extra overhead in the common case in which a currency only issues soft tickets, while still allowing hard tickets to be issued by any currency. Users could use hard tickets to give an application a fixed percentage of their resource rights, or to specify hierarchical reservations in which absolute shares from the base currency are divided into hard subshares. For a hard ticket to represent a fixed-share reservation of the actual resource, the path from the root currency to the client must involve only hard tickets.

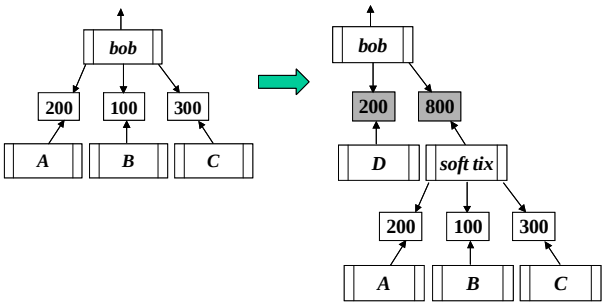

Figure 4: Offering Hard Shares of a Currency's Resource Rights. The *bob* currency issues a hard ticket to task D representing a fixed 20% (200/1000) of *bob*'s resource rights. As a result, a special subcurrency (*soft_tix*) is created and used to fund *bob*'s soft tickets and isolate the hard tickets from changes in the number of soft tickets. The funding given to the *soft_tix* subcurrency is adjusted as needed to ensure that the total number of hard tickets issued by *bob* remains fixed at 1000.

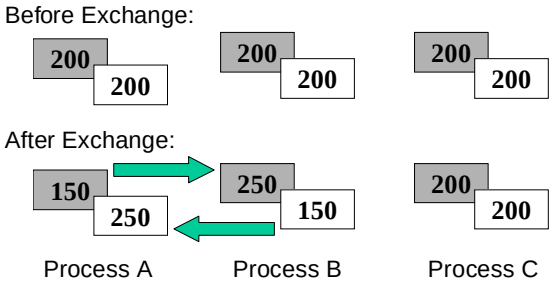

Figure 5: Ticket Exchanges Insulate Non-Participants. Processes A and B exchange tickets. Process C remains unaffected, since it still has one-third of the total of each ticket type.

4 Isolation with Greater Flexibility

We now describe the mechanisms that we have developed to safely overcome the limits that currencies impose. These mechanisms allow our extended framework to provide isolation with greater flexibility, enabling clients to obtain resource allocations that better meet their differing and dynamically changing needs.

4.1 Ticket Exchanges

As discussed in Section 2.2.1, a currency imposes an upper limit on the resource allocations that clients funded by its tickets can receive. However, these limits may often be unnecessarily restrictive. This is especially true on central servers, since the large number of clients that a server must accommodate makes it difficult for a single allocation policy to adequately address clients' different and dynamically changing resource needs. Instead, some simple policy for ensuring fairness is likely to be used, such as giving users equal resource rights to divide among their applications or allocating resource shares based on how much a user has paid.

Since certain resources may be more crucial than others to the performance of an application, applications may benefit from giving up a fraction of their default allocation for one resource in order to receive a larger share of an another resource. We have therefore developed a mechanism called *ticket exchanges* that allows applications to take advantage of their differing resource needs by bartering with each other over resource-specific tickets. For example, a CPU-intensive application could exchange some of its disk tickets for some of the CPU tickets of an I/O-intensive application.

While ticket exchanges allow clients to obtain additional resource rights, they do so without compromising the isolation properties of the lottery-scheduling framework. As the scenario depicted in Figure 5 illustrates, only the resource rights of clients participating in an exchange are affected by it; the resource rights of non-participants are unaffected.

Ticket exchanges are not, however, guaranteed to preserve the resource *shares* that non-participants actually received before the exchange occurred. Since each of the clients involved in an exchange will presumably make greater use of the resource for which it obtains extra tickets than the client who traded the tickets away, there will likely be increased contention for the resources involved. As a result, other clients who previously received larger resource shares than their resource rights guaranteed may now see those shares reduced. For example, if a CPU-intensive client trades some of its disk tickets to a client that regularly accesses the disk, those previously inactive disk tickets will suddenly become active, and the disk tickets of other clients accessing the disk will now be worth less. However, all clients will still receive at least the minimal shares to which their tickets entitle them.

Ticket exchanges and currencies complement each other. Exchanges allow for greater flexibility in the face of the upper limits on resource allocation imposed by currencies, while currencies insulate processes from the malicious use of exchanges. For example, a process could fork off children that use exchanges to give the process all of their tickets. With currencies, however, this tactic would only affect the resource rights of tasks funded by the same currency as the malicious process.

4.2 Determining and Coordinating Exchanges

Ticket exchanges enable applications to coordinate with each other in ways that are mutually beneficial and that may increase the overall efficiency of the system. Various levels of sophistication could be employed by applications to determine what types of exchanges they are willing to

make, and at what rates of exchange.

Certain types of clients may primarily need extra tickets for one particular resource. For example, consider two Web sites that are virtually hosted on the same server. Site A has a small number of frequently accessed files that it could keep in memory if it had additional memory tickets for its currency. Site B has a uniformly accessed working set that is too large to fit in memory, and it would thus benefit from giving up some of its currency's memory tickets for some of A's disk tickets.

Clients could also apply economic and decision-theoretic models to determine, based on information about their performance (such as how often they are scheduled per second, how many page faults they incur per second, etc.) and the current state of the system (such as how many tickets of each type are active in the system), when to initiate an exchange and at what rate. This determination could be made by the client process itself, or by a separate *resource negotiator* process that monitors the relevant variables and initiates exchanges on the client's behalf. Resource negotiators are similar to the *application mangers* proposed by Waldspurger [Wal95], who suggested funding these helper processes using a small percentage of the funding of their applications.

Clients are free to cancel exchanges in which they are involved. This allows clients to take a trial-and-error approach, experimenting with a variety of exchange rates until they achieve an acceptable level of performance, and to adapt their resource usage over time.

Clients or their negotiators initiate exchanges by sending the appropriate information to a central *dealer*. The dealer maintains queues of outstanding exchange proposals, attempts to match up complementary requests, and carries out the resulting exchanges. If an exchange request cannot be immediately satisfied, the dealer returns a message that includes any proposals with conflicting exchange rates (e.g., process A requests 20 CPU tickets for 10 memory tickets, while process B requests 10 memory tickets for 10 CPU tickets). In this way, clients or their negotiators can decide whether to modify their proposed exchange rate and try again for a compromise deal. In environments where isolation is less important, the dealer could be modified to carry out exchanges that processes propose on the processes themselves (e.g., to take away 20 CPU tickets from a process and give it 20 extra memory tickets in return), giving an approach equivalent to the one suggested by Waldspurger (see Sect. 3.1).

Future research is needed to develop negotiators suitable for a wide variety of applications and environments. Among the questions that still need to be addressed are: How can a negotiator determine what exchanges are beneficial to its associated process? When should a negotiator accept a trade less desirable than the one it proposed? Will a system involving dynamic ticket exchanges be stable (i.e., how can oscillatory behavior be avoided)? Can general-purpose negotiators be written that avoid the need to craft one for each application? In addition, the central dealer must be designed to deal fairly with requests that have complementary but differing exchange rates.

4.3 Carrying Out an Exchange

Once a complementary set of exchange requests is found, the funding of the clients involved must be modified to reflect the exchange. In a non-hierarchical system with no subcurrencies, this could be accomplished by directly modifying the number of tickets that the clients hold for the resources involved. However, the presence of subcurrencies complicates matters, because the base value of subcurrency tickets changes over time, whereas tickets involved in exchanges should have a constant base value.

To address this problem, exchanges are accomplished by issuing four ticket objects directly from the base currency: two for the amounts being exchanged, and two *negatively valued* tickets for the opposite amounts. For example, if client A trades some of its disk tickets with a base value of 50 for some of client B's memory tickets with a base value of 20, then A is given 20 memory tickets and −50 disk tickets, and B is given 50 disk tickets and −20 memory tickets. Since scheduling and allocation decisions involve computing the total base value of a client's tickets for a particular resource, the negative tickets will reduce the exchanging clients' resource rights by the amount that they have traded away.

An exchange is undone if one of the exchanging clients exits, cancels the exchange, or loses the resource rights that it traded away. This latter scenario can occur if the tickets funding the client decrease in value to the point that their total value becomes less than the value of the tickets that the client gave away. In such cases, the client's total base value for that resource becomes negative, and the exchange must be revoked.

4.4 Enabling the Semantics of Nice

While ticket exchanges provide greater flexibility in the face of the upper limits enforced by currencies, we also need to overcome the lower limits that currencies impose by providing a mechanism for renicing resource-intensive jobs. As noted in Section 2.2.2, such jobs would ideally be given a small amount of funding directly from the base currency, but unprivileged users would not ordinarily be allowed to issue base-currency tickets.

To circumvent this restriction, the system can employ a policy allowing users to issue base-currency tickets provided that the total value of the tickets funding currencies owned by the user never exceeds some upper bound. In

this way, users can take away a small amount of their currencies' funding, and then issue that same amount from the base currency to fund resource-intensive jobs. This approach leaves the user's total resource rights unchanged (preserving the isolation of other users), and the new job can run at a reduced priority without crippling the user's other applications. We discuss the details of our implementation of this policy in Section 5.6.

5 Prototype Implementation

To test our ideas, we implemented the extended lottery-scheduling framework in VINO 0.50 (www.eecs.harvard.edu/~vino/vino), and we use it to manage CPU time, memory, and disk bandwidth. In the following sections, we describe the key details of our implementation, including the scheduling and allocation mechanisms we employ.

5.1 Pure and Client Currencies

In our implementation, threads are themselves currencies, since they hold backing tickets and may even need to issue tickets as part of a temporary *ticket transfer* [Wal94] to one or more other processes on which they are waiting (see Sect. 5.3). Since VINO is written in C++, we implement the thread-currency duality by having VINO's thread class inherit from our currency class. Currencies that are also threads are referred to as *client currencies* and are identified by the process id of the thread. All other currencies are referred to as *pure currencies*, and are identified by a unique currency identifier (cid).

5.2 Currency Configuration and Access Controls

By default, the system creates one currency for each active user of the system, and it funds these *user currencies* equally from the base currency. Each user's currency in turn funds the processes of that user. The user currencies are created by means of a function, cid_for_client(), that is invoked when a process changes its real user id (uid); this function uses a uid to cid mapping to determine which currency should be used to fund the process. If no mapping exists for a given user, a new currency is created and funded. In either case, the process' existing funding is revoked and replaced with funding from the appropriate user currency. Once a user's login shell is correctly funded, processes forked by that shell are funded by the same currency. More generally, child processes are funded by the issuer of the first ticket in the parent's list of backing tickets for that resource.[3] When a process is re-funded by a different user currency, the new funding tickets are placed at the head of the process' backing-ticket lists so that its children will be properly funded.

Each user currency is owned by the corresponding user and has a currency mode (see Sect. 3.2) that allows the user to manipulate it using a set of system calls that we added for this purpose. For example, a user's processes receive equal ticket allocations by default; the user can modify these allocations and thus alter the relative resource rights that the processes receive. Users can also create new currencies and fund them with tickets from their user currency. However, they cannot increase the funding of their user currency itself, since they do not typically have permission to issue tickets from other currencies. Therefore, each user's tasks are securely isolated from the tasks of other users, and the total resource rights of each user are identical.

Other currency-configuration policies could also be specified, such as one currency per group or some combination of user and group, along with different levels of funding for different clients. On extensible operating systems like VINO [Sma98], superusers could safely download specialized versions of the cid_for_client function (which is also called when a process' real group id (gid) changes) to specify arbitrary configuration schemes based on uid and gid, as well as alternative currency access-control policies. Approaches that do not involve extensibility could also be employed to accommodate a more limited range of possible configurations and access controls.

5.3 Managing CPU Time

Our current implementation schedules the CPU using the original lottery-scheduling algorithm [Wal94], randomly choosing an active CPU ticket and traversing the runnable queue to find the thread holding that ticket. To find the winning thread, the system computes the base value of each thread's CPU backing tickets, tracing up the currency graph and converting ticket values until it determines their values with respect to the base currency. We cache these base values to avoid unnecessary computation, although the cached values must be invalidated whenever a change occurs in a currency's count of active tickets (e.g., when a thread starts up, blocks, or exits).

Our prototype also uses two other features of the original lottery-scheduling framework, *compensation tickets* and *ticket transfers*. Compensation tickets are issued to clients who do not use their full quantum, temporarily inflating their resource rights so that they will be chosen more quickly when they next become runnable; they thereby restore the clients' expected resource shares. Ticket transfers occur when a client blocks attempting to acquire a kernel mutex or to allocate memory. Since the client is itself a currency (see Sect. 5.1), it issues tickets and uses

3. Actually, if a client holds tickets from more than one pure currency, its children should be funded by all of these currencies. We plan to extend our implementation to deal with this case.

them to fund the thread on which it is waiting (the holder of the mutex or the pageout daemon), transferring its resource rights to that thread. This can reduce the time that the client spends blocked, and it prevents priority inversion from occurring. When the client is made runnable again, its transfer tickets are revoked.

The lottery-scheduling algorithm has a number of advantages; in particular, it requires minimal per-client state and it easily accommodates dynamic changes to the set of runnable processes. However, it only provides probabilistic guarantees, and can thus result in poor throughput accuracy over short time intervals. A number of deterministic algorithms can also be used within the lottery-scheduling framework to provide increased accuracy and lower response-time variability for interactive processes, including stride scheduling [Wal95] and EEVDF [Sto96]. However, Waldspurger [Wal95] suggests that lottery scheduling's randomization may be effective in preventing malicious clients from exploiting the system's resources, as well as in providing greater stability in systems that use the lottery-scheduling framework to manage multiple resources. Since our work primarily addresses ways of adding flexibility to a proportional-share framework, the algorithms used are not crucial.

A system designed for central servers should properly account for kernel-mode processing done on a client's behalf [Dru96, Ban99]. This is especially important when it comes to handling incoming network traffic, which is typically done by kernel threads. When we extend our prototype to manage network bandwidth, we intend to add support to properly account for this processing.

5.4 Managing Memory

Effective proportional-share memory management is complicated by the difficulty of determining which clients are actively competing for memory and by the undesirability of a strict partitioning of memory among the clients. When scheduling the CPU, threads that are blocked are simply ignored and the values of their tickets are not counted. Our data indicates that a similar approach to memory management is not effective, because when the system experiences heavy memory pressure, any process that blocks, even momentarily, will potentially lose a large number of pages to the activity of the pageout daemon, resulting in erratic paging behavior and poor throughput (Figure 6). The obvious alternative, namely leaving the memory tickets of all processes active at all times, is also not viable, since pages belonging to idle processes tend to remain in memory indefinitely, reducing the number of pages available to active processes and effectively partitioning the total memory of the system. We have therefore chosen to issue memory tickets only to clients that explicitly request them. Processes running as root are allowed to obtain hard

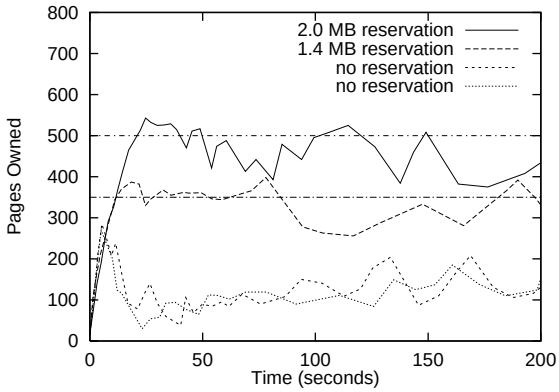

Figure 6. A system in which memory tickets are deactivated whenever a client blocks. Four memory-intensive tasks compete for approximately 4.0 MB (1024 pages) of available memory. Processes with hard shares ("reservations") are unable to maintain those shares due to the pages that are reclaimed when they block.

memory shares from the base currency, specified in kilobytes. Once a currency is funded with memory tickets, clients with appropriate permissions can obtain soft or hard subshares of that currency's allocation.

VINO approximates an LRU page-replacement policy by maintaining two lists, one of pages considered active and one of pages considered inactive. The pageout daemon periodically moves unreferenced pages from the active list to the inactive list, and it reclaims pages from the inactive list to maintain a desired number of free pages. Pages on the inactive list that have been referenced are returned to the active list rather than being reclaimed. To maintain guaranteed memory shares, we altered the behavior of VINO's pageout daemon so that pages owned by clients who have not exceeded their memory share are also not reclaimed. This approach does not limit clients to their memory shares, but merely guarantees that they can receive at least that amount. In the absence of memory pressure, clients receive as much memory as they need.

If a client holds soft memory tickets, the number of pages to which it is entitled can change dynamically as the value of its tickets changes. The pageout daemon thus needs to compute the current base value of the client's memory tickets; cached values are used when possible. A more complete implementation would probably benefit from deactivating the memory tickets of clients who are idle for a sufficiently long period of time. Since most clients do not receive memory tickets under our current approach to memory management, our prototype does not attempt to do this.

Other approaches to proportional-share memory management are also possible, including the *inverse lottery* and *min-funding revocation* schemes developed as part of the original lottery-scheduling work [Wal95]. Both of

these schemes involve first choosing a client whose page should be revoked, and then selecting one of its pages; this involves potentially recomputing each client's base value on every revocation decision and then traversing the LRU list to find one of its pages. We were concerned about the potential overhead of these schemes, and thus adopted the simpler approach described above. Ultimately, we are interested in investigating an alternative approach in which all processes receive memory tickets. Under this approach, processes' memory allocations would be dynamically adjusted so that their time-averaged page-fault rates were proportional to the value of their memory tickets.

5.5 Managing Disk Bandwidth

Our prototype supports proportional sharing of disk bandwidth using the hierarchical YFQ algorithm developed by Bruno et al. [Bru99b]. YFQ is similar to weighted-fair-queueing schemes for managing network bandwidth [Par93, Ben96]. It approximates ideal proportional sharing by maintaining a virtual time function and per-disk queues of outstanding disk requests for each client. Each client queue has an associated *finish tag* that reflects the client's past disk activity, its current share of the disk, and the length of the request at the head of the queue. Requests from queues with the smallest finish tags are forwarded to the device driver in small batches that can be reordered by the driver or device to achieve better aggregate throughput. Bruno et al. describe other techniques for increasing throughput while maintaining close to the desired proportional shares. We have not yet implemented these optimizations, although we do bypass the client disk queues if there is only one active client with no enqueued requests and the number of its outstanding requests is smaller than the size of the batches sent to the disk.

A client's disk tickets are active whenever they have an outstanding request. To adjust to dynamic changes in the number of active disk tickets, the base value of a client's disk tickets is recomputed (using cached values if possible) whenever a request reaches the head of its client's queue, and this value is used to compute the queue's new finish tag.

When a client blocks attempting to acquire a kernel mutex or to allocate memory, it temporarily transfers its disk tickets along with its CPU tickets (see Sect. 5.3) to the client on which it is waiting. The system, when forwarding queued disk requests to the device driver, skips over requests from clients who have transferred their disk tickets, unless there are no other requests available.

5.6 Emulating Nice

To support the semantics of nice, we created a utility that runs with root privileges and can be used to execute programs with funding from the base currency. This utility reduces the funding of the caller's user currency by the amount requested for the job being niced, thus preserving isolation. This utility actually creates a new currency for the job, funds that currency with the requested number of tickets, and uses the new currency's tickets to fund the job. This level of indirection is needed in case the job spawns any children; if so, they will share the funding originally given to the job. While this utility would typically be used to specify a small amount of CPU funding for a long-running CPU-intensive job, it can be used with other resources as well. This simple utility successfully overcomes the lower limits imposed by currencies without employing VINO's extensibility mechanism, as we had originally planned [Sul99]. The other broker methods could also be overridden using similar setuid-root utilities.

It is important to note that this solution would not work under all currency-configuration policies. For example, if a system uses one currency per group rather than one currency per user, it would probably not be appropriate to take away funding from the group currency to fund a user's resource-intensive job; in fact, doing so could end up counteracting the intended benefits of "nice-ing" the job. Under these alternate currency configurations, it might make sense to use a more general approach in which negative tickets issued from the base currency are used to directly reduce the funding of the resource-intensive job (cf. our method for implementing exchanges in Sect. 4.3). The complication with this approach is that the number of negative tickets used would need to be adjusted to accommodate variations in the number of other active tasks funded by the same currency.

5.7 Carrying Out Exchanges

As discussed in Section 4.2, a number of challenging questions must be answered before a system that fully supports dynamic ticket exchanges can be built. At this point, we have implemented a framework that allows us to easily test the effects of ticket exchanges and thereby gain insight into the issues involved.

We provide two mechanisms for experimenting with exchanges. First, user-level utilities that employ our mkcur(), fund(), and unfund() system calls can be used by users with appropriate permissions to implement "static" exchanges, preset modifications to the default ticket allocations. Second, we have temporarily moved the role of the central dealer into the kernel, and we allow applications to propose exchanges dynamically using a system call that we added for this purpose. When an exchange is

carried out, the four tickets involved (see Sect. 4.3) are linked to each other in a circular queue so that the exchange can be invalidated when one of the clients exits, loses too much funding, or decides to retract the exchange. If one of the tickets in the queue is deleted, all four of them are, thereby cancelling the exchange.

6 Experiments

We conducted a number of experiments to test the effectiveness of our extended framework and to determine how well it succeeds in providing greater flexibility while preserving secure isolation. In the following sections, we first discuss tests of the proportional-share mechanisms that we implemented and demonstrate that they provide accurate proportional-share guarantees and effective isolation. We then present experiments that test the impact of ticket exchanges on two sets of applications.

6.1 Experimental Setup

All of these experiments were conducted using our modified kernel. We ran it on a 200-MHz Pentium Pro processor with 128 MB of RAM and a 256-KB L2 cache. The machine had an Adaptec AHA-2940 SCSI controller with a single 2.14-GB Conner CFP2105S hard disk.

6.2 Providing Shares of CPU Time

To test the effectiveness of our implementation of the basic lottery-scheduling framework, including the lottery algorithm for CPU scheduling, we replicated an experiment from the original lottery-scheduling paper (Wal94, Sect. 5.5). We ran five concurrent instances of a CPU-intensive program (the Dhrystone benchmark [Wei84]) for 200 seconds using the CPU funding shown in Figure 7. Client B3 sleeps for the first half of the test, during which time its tickets are not active.

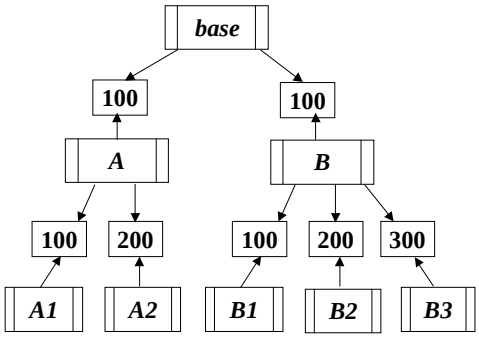

Figure 7: CPU funding hierarchy. This figure depicts the funding used for the experiment described in Section 6.2. Currencies A and B receive equal funding from the base currency, which they divide among the tasks they fund. A2 receives twice the funding of A1, B2 receives twice the funding of B1, and B3 has three times the funding of B1.

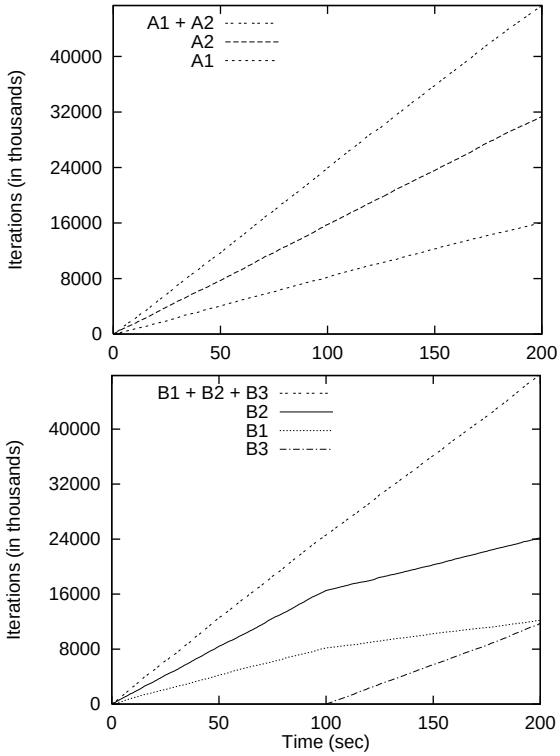

Figure 8: Hierarchical Proportional Sharing of CPU Time. Five CPU-intensive tasks, with funding shown in Figure 7, compete for the CPU. Shown above are the number of iterations completed by each task as a function of time. Task B3 sleeps for the first half of the test.

Figure 8 shows the number of iterations accomplished as a function of time for the clients funded by each currency. In all cases, the relative levels of progress of the clients match their relative funding levels. When B3 awakens, its tickets are reactivated; as a result, the other clients funded by currency B receive a reduced share of the CPU, while the clients funded by currency A are unaffected because of the isolation that currencies provide.

6.3 Providing Memory Shares

The next experiment tests our prototype's ability to guarantee fixed shares of physical memory. To create enough memory pressure to force frequent page reclamation, we limited the size of accessible memory to 8 MB. After subtracting out the pages wired by the kernel as well as the desired number of free pages in the system, there were approximately 4.2 MB of memory that clients could reserve. We ran four concurrent instances of a memory-intensive benchmark; each instance repeatedly reads random 4-KB portions of a 16-MB file into random locations in a 4-MB buffer. This load keeps the pageout daemon running more or less continuously. We gave one client a 2-MB hard memory share (500 pages) and another

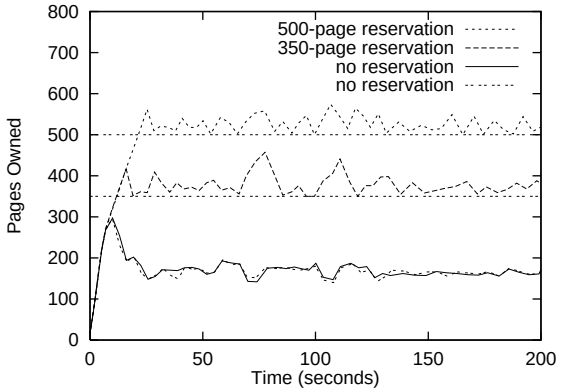

Figure 9: Providing Hard Memory Shares. Four memory-intensive tasks run concurrently on a system with approximately 4.2 MB of available memory. Two have guaranteed memory shares; two do not. Shown are the number of 4-KB pages owned by each process as a function of time.

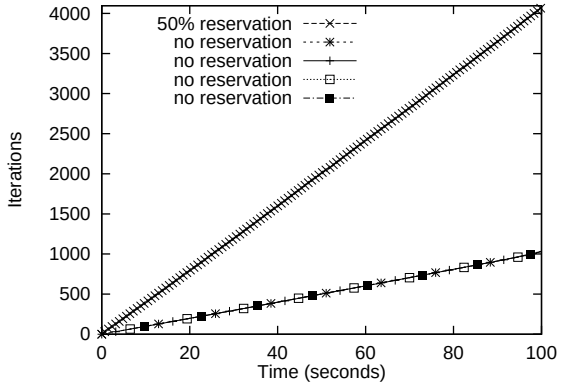

Figure 10: Providing Proportional Shares of Disk Bandwidth. Five I/O-intensive tasks compete for access to the disk. One of them receives a 50% hard share, while the others receive equal funding from their user's currency and thus divide up the other half of the available bandwidth. Each iteration corresponds to paging in one 4-KB page of a memory-mapped file. The lines for the four tasks with soft shares overlap almost perfectly.

client a 1.4-MB hard memory share (350 pages); the other two clients were not guaranteed any memory. Figure 9 shows the number of pages owned by each client as a function of time. The clients with hard shares lose pages only when they own more than their guaranteed share. The clients without memory tickets have no guaranteed share and end up with much less memory than the other two.

6.4 Providing Shares of Disk Bandwidth

We tested our implementation of the YFQ algorithm for proportional-share disk scheduling by running five concurrent instances of an I/O-intensive benchmark (iohog) that maps a 16-MB file into its address space and touches the first byte of each page, causing all of the pages to be brought in from disk; each copy of the benchmark used a different file. Throughout the test, each client almost always has one outstanding disk request. We set YFQ's batch size to 1 so that it would provide strict proportional sharing. We gave one client a 50% hard share of the disk (i.e., one-half of the base currency's hard disk tickets), while the other four clients received the default number of disk tickets from their user's currency. Figure 10 shows the number of iterations that each client accomplishes over the first 115 seconds of the test. Since one client has reserved half of the disk, the other four clients divide up the remaining bandwidth and effectively get a one-eighth share each. Thus, the client with the hard share makes four times as much progress as the others; when it has finished touching all 4096 of its file's pages, the other four have touched approximately 1000 pages (a 4.1:1 ratio).

6.5 Ticket Exchanges: CPU and Disk Tickets

To study the impact of ticket exchanges, we first conducted experiments involving the CPU-intensive dhrystone benchmark [Wei84] and the I/O-intensive iohog benchmark (see Sect. 6.4). In the first set of runs, we gave the benchmarks allocations of 1000 CPU and 1000 disk tickets from the base currency. We then experimented with a series of one-for-one exchanges in which dhrystone gives iohog n disk tickets in exchange for n CPU tickets, where n = 100, 200, …, 800. To create added competition for the resources—as would typically be the case on a central server—we ran additional tasks (one extra dhrystone and four extra iohogs) in the background during each experiment. Each of the extra tasks received the standard funding of 1000 CPU and 1000 disk tickets.

Figure 11 shows the performance improvements of the exchanging applications under each exchange, in comparison to their performance under the original, equal allocations. Dhrystone benefits from all of the exchanges, and the degree of its improvement increases as it receives additional CPU tickets. Iohog also benefits from all of the exchanges, but the degree of its improvement decreases for exchanges involving more than 500 tickets. While dhrystone does almost no I/O and can thus afford to give up a large number of disk tickets, iohog needs to be scheduled in order to make progress, and thus the benefit of extra disk tickets is gradually offset by the loss of CPU tickets. However, both applications can clearly benefit from this type of exchange, which takes advantage of their differing resource needs.

We also examined the effect of the ticket exchanges on the non-exchanging tasks. As discussed in Section 4.1,

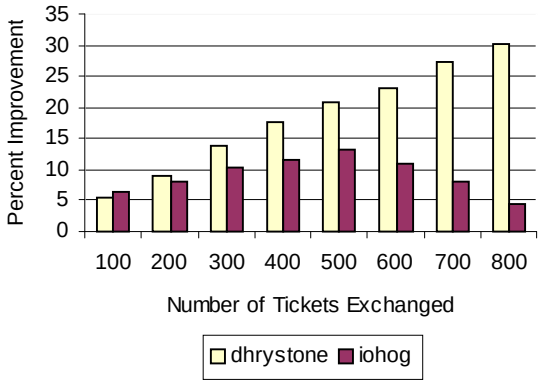

Figure 11: Performance Improvements from Ticket Exchanges. A CPU-intensive task (*dhrystone*) exchanges disk tickets for some of the CPU tickets of an I/O-intensive task (*iohog*). The improvements are with respect to runs in which the tasks receive the default ticket allocations. All results are averages of five runs.

the resource *rights* of these tasks should be preserved, but their actual resource *shares* may be affected. Such an effect is especially likely in these experiments, because the two benchmarks rely so heavily on different resources. For example, dhrystone uses almost no disk bandwidth. As a result, the iohogs obtain more bandwidth than they would if the dhrystones were competing for the disk. However, when the exchanging iohog receives some of the exchanging dhrystone's disk tickets, it obtains rights to a portion of this "extra" bandwidth, and thus the other iohogs end up with smaller bandwidth shares. Exchanges affect the CPU share of the non-exchanging dhrystone in the same way.

However, the non-exchanging processes should still obtain at least the resource shares that they would receive if all of the other tasks were continuously competing for both resources. To verify this, we used getrusage (2) to determine each task's CPU and disk bandwidth usage during the first 100 seconds of each run. The results (Fig. 12) show that the minimal resource rights of the non-exchanging processes are preserved by all of the exchanges. The top graph shows the CPU shares of both the exchanging and non-exchanging dhrystones, and the bottom graph shows the disk-bandwidth shares of the exchanging and non-exchanging iohogs[4]. Since there are seven tasks running during each test, the non-exchanging tasks are each guaranteed a 1/7 share (approximately 14.3%). The non-exchanging iohogs are affected less than the non-exchanging dhrystone because each of them loses only a portion of the bandwidth gained by the exchanging iohog. In general, as the number of tasks competing for a resource increases, the effect of exchanges on non-exchanging tasks decreases.

4. The four non-exchanging iohogs have approximately equal shares. In each case, the graphed value is the smallest share of the four.

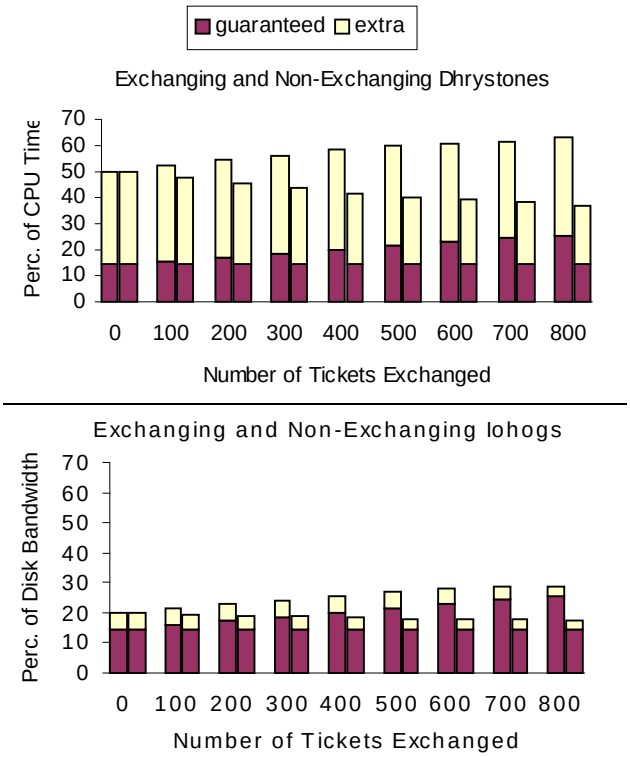

Figure 12: Resource Shares under Exchanges. Shown are the CPU shares of the exchanging and non-exchanging dhrystones (*top*) and the disk-bandwidth shares of the exchanging and non-exchanging iohogs (*bottom*). The dark portion of each bar represents the share guaranteed by the task's tickets, while the full bar indicates its actual share. In each pair, the left bar is the exchanging copy, and the right bar is the non-exchanging copy. All results are averages of five runs.

6.6 Ticket Exchanges Between Database Applications: Memory and Disk Tickets

We further experimented with ticket exchanges using two simple database applications that we developed using the Berkeley DB package [Ols99]. Both applications emulate a phone-number lookup server that takes a query and returns a number; when run in automatic mode, they repeatedly generate random queries and service them. One of the applications (*small*) has a 4-MB database with 70,000 entries, while the other (*big*) has a much larger, 64-MB database with 2^{20} entries. Both applications use a memory-mapped file as a cache.

We ran these applications concurrently for a series of 300-second runs. We disabled the update thread for the sake of consistency, because its periodic flushing of dirty blocks from the applications' cache files can cause large performance variations. To emulate the environment on a busy server, we created added memory pressure—limiting the available memory to 16 MB—and we ran four iohogs in the background. After subtracting out the pages wired

by the kernel and the system's free-page target, there was approximately 11.1 MB that clients could reserve. When *small* runs alone, it ends up using up to 8 MB of memory as a result of double buffering between the filesystem's buffer cache and its own 4-MB cache. With only 70,000 entries, it makes a large number of repeated queries, and it should thus benefit from additional memory tickets that allow it to cache more of its database. On the other hand, *big* only uses a 500-KB cache because it seldom repeats a query; it should benefit from additional disk tickets.

We started by giving the applications equal allocations for each resource: 1000 CPU tickets, 1375 hard memory tickets[5], and 1000 disk tickets, all from the base currency. We then experimented with exchanges in which *small* gives up some of its disk tickets for some of *big*'s memory tickets, trying all possible pairs of values from the following set of exchange amounts: {100, 200, …, 800}. The iohogs always had 1000 CPU and 1000 disk tickets each.

While the exchanges in Section 6.5 were preset, the exchanges in these experiments were proposed and carried out dynamically using the exch_offer() system call (see Sect. 5.7). *Big* proposes the exchange as soon as it starts running, but *small* waits until it has made 20,000 queries (approximately halfway through the run), at which point the exchange is carried out. By waiting, *small* is able to use its original disk-ticket allocation to bring a portion of its database into memory quickly, at which point it can afford to exchange some disk tickets for memory tickets.

S*mall* benefits from most of the exchanges, including any exchange in which it obtains 400 or more memory tickets. It fails to benefit when it gains only 100 memory tickets, or when it gives away a large number of disk tickets for 300 or fewer memory tickets (Fig. 11a-c). Since *small* can fit no more than three-quarters of its database in memory with these memory allocation, it cannot afford to give away a large number of disk tickets. When *small* obtains 700 or 800 memory tickets, it can hold all of its database in memory, and it thus sees performance improvements of over 1000 percent (Fig. 11g, h). *Big* likewise benefits from most of the exchanges, including any exchange in which it obtains 600 or more disk tickets.

It is interesting to note that these applications cannot simply specify an exchange *ratio*, such as two disk tickets for every one memory ticket, since what constitutes an acceptable ratio depends on the number of tickets being exchanged. For example, *small* should not accept a ratio of 2:1 disk:memory if only 100 or 200 memory tickets are offered, but it should accept exchanges with this ratio to obtain 300 or more memory tickets. More generally, what constitutes an acceptable exchange depends heavily on the environment in which the tasks are running. For example, because a task needs to wait until a synchronous I/O completes before enqueueing a new one, tasks can receive at most 50% of the bandwidth in the absence of prefetching. Therefore, without extra tasks competing for the disk, *big* cannot benefit from extra disk tickets, because it already obtains 50% of the disk by default. Applications like *big* will need to use negotiators that can assess the current system conditions before proposing an exchange.

5. When receiving a hard memory ticket from the base currency, the value of the ticket actually represents the number of pages in the associated memory reservation. In this case, 1375 pages corresponds to just under half of the available physical memory.

7 Related Work

Other systems have allowed applications to negotiate their resource usage with the operating system [Jon95, Nob97]. The resource-management system proposed for the Rialto OS [Jon95] allows applications to negotiate for needed resources with a local "resource planner." Applications renegotiate with the resource planner as their needs change, and make trade-offs among resources if their requested allocations cannot be met. We hope to develop resource negotiators that will be able to perform the types of self-monitoring, reasoning about resources, and performance-tuning that Rialto requires. Our system's central dealer will serve some of the same functions as the Rialto resource planner.

In the Odyssey system for mobile computing [Nob97], the system monitors changes in resource availability, notifies applications of relevant changes, and allows them to decide how best to adapt. Our system is similar in spirit to their "application-aware" approach, leaving adaptation to applications while using the system to control and enforce resource allocations.

Our extended lottery-scheduling framework differs from existing ones in the ability it gives applications to coordinate their resource usage with *each other*, as well as with the system as a whole.

Much of the recent work on resource management has been motivated by the need to accommodate soft real-time (i.e., multimedia) applications [Bru98, Bru99, Jon97, Nie97]. Our hard-ticket mechanism allows lottery scheduling to support the resource reservations that such applications require.

There has been a good deal of work on proportional-share scheduling of single resources, primarily CPU time [Mer94, Goy96] or network bandwidth [Par93, Ben96]. Our extended lottery-scheduling framework securely manages multiple resources. In Section 2.3, we discuss several other resource-management frameworks that also support proportional-sharing of multiple resources, and we compare them to the lottery-scheduling framework. One of these frameworks, the Software Perfor-

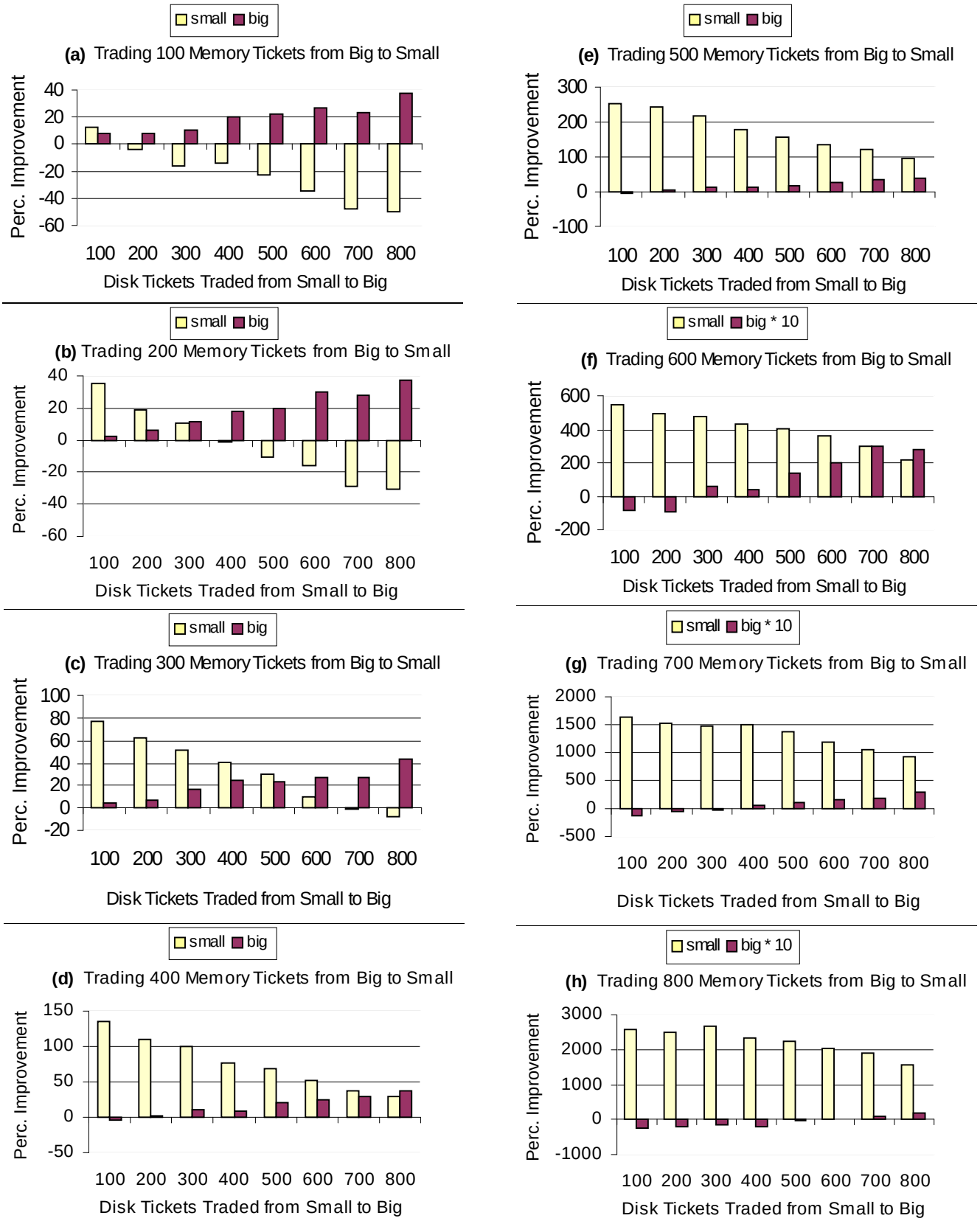

Figure 11: A variety of exchanges in which a database application with a large working set (*big*) exchanges memory tickets for some of the disk tickets of a similar application with a small working set (*small*). The graphed changes compare the number of requests serviced in a 100-second interval after the exchange has occurred with the requests serviced during the same interval when no exchange is made. All results are averages of at least five runs. Note that there are different vertical scales, and the values for *big* in graphs (f), (g) and (h) are multiplied by 10 to make them more visible.

mance Units (SPUs) of Verghese et al. [Ver98], explicitly addresses the need to provide both isolation and flexibility. However, their system starts by giving absolute resource shares to each SPU, and it gains added flexibility by dividing unused portions of these shares among SPUs that need additional resources. The original lottery-scheduling framework naturally supports this type of resource sharing by deactivating the tickets of idle clients. Our extended framework provides added flexibility through ticket exchanges, and through the utility that we use to emulate the semantics of nice.

Besides Waldspurger's own prototypes, others have implemented portions of the lottery-scheduling framework [Arp97, Nie97]. In particular, Petrou et al. [Pet99] retrofitted lottery scheduling into FreeBSD to schedule the CPU, extending the framework to better support interactive jobs. VINO currently has a small, 10-ms quantum (as compared to FreeBSD's 100-ms one), so such extensions have not been needed in our prototype. Petrou et al. also suggest an alternative approach to overcoming the lower limits that currencies impose.

8 Conclusions

Our extended lottery-scheduling resource-management framework provides a means for both flexible responsiveness to the differing needs of applications and isolation between groups of processes. We believe that its ability to provide isolation with greater flexibility makes it an excellent choice for systems in which many users compete for the resources of a central server, as in thin-client networks or Web servers used for virtual hosting. Ticket exchanges allow processes to adjust their ticket allocations while insulating clients that do not take part in the exchange, and they provide a means for applications to coordinate their resource usage with each other. Currency brokers provide secure access controls to currencies, while setuid utilities can be used to flexibly circumvent the default controls in ways that preserve isolation.

In order for our extended framework to be fully effective on large central servers, more work needs to be done to develop negotiators that can intelligently carry out ticket exchanges on behalf of users and applications. Developing such negotiators will be a challenging task, but one with potentially significant rewards.

Acknowledgments

This research was supported in part by a USENIX Association Scholarship. We thank Robert Haas for contributing to the development of our extended lottery-scheduling framework, and Carl Waldspurger for helpful comments on earlier drafts.

References

[Arp97] Arpaci-Dusseau, A.C., Culler, D.E., "Extending Proportional-Share Scheduling to a Network of Workstations," *Proc. of the Int'l Conf. on Parallel and Distributed Processing Techniques and Applications*, June 1997.

[Ban99] Banga, G., Druschel, P., Mogul, J.C., "Resource Containers: A New Facility for Resource Management in Server Systems," *Proc. of the Third Symposium on Operating Systems Design and Implementation*, February 1999.

[Ben96] Bennett, J., Zhang, H., "Hierarchical Packet Fair Queueing Algorithms," *Proceedings of IEEE SIGCOMM '96*, August 1996.

[Bru98] Bruno, J., Gabber, E., Özden, B., Silberschatz, A., "The Eclipse Operating System: Providing Quality of Service via Reservation Domains," *Proc. of the USENIX 1998 Annual Technical Conference*, June 1998.

[Bru99a] Bruno, J., Brustoloni, J., Gabber, E., Özden, B., Silberschatz, A., "Retrofitting Quality of Service into a Time-Sharing Operating System," *Proc. of the USENIX 1999 Annual Technical Conference*, June 1999.

[Bru99b] Bruno, J., Brustoloni, J., Gabber, E., Özden, B., Silberschatz, A., "Disk Scheduling with Quality of Service Guarantees," *Proc. of the Int'l Conference on Multimedia Computing and Systems*, June 1999.

[Dru96] Druschel, P., Banga, G., "Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems," *Proc. of the Second Symposium on Operating Systems Design and Implementation*, October 1996.

[Goy96] Goyal, P., Guo, X., Vin, H.M., "A Hierarchical CPU Scheduler for Multimedia Operating Systems," *Proc. of the Second Symposium on Operating Systems Design and Implementation*, October 1996.

[Jon95] Jones, M.B., Leach, P.J., Draves, R.P., Barrera, J.S., "Modular Real-Time Resource Management in the Rialto Operating System," *Proc. of the Fifth Workshop on Hot Topics in Operating Systems*, May 1995.

[Jon97] Jones, M.B., Rosu, D., Rosu, M-C., "CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities," *Proc. of the 16th ACM Symposium on Operating System Principles*, October 1997.

[Mer94] Mercer, C.W., Savage, S., Tokuda, H., "Processor Capacity Reserves for Multimedia Operating Systems," *Proc. of the IEEE Int'l Conference on Multimedia Computing and Systems*, May 1994.

[Nie97] Nieh, J., Lam, M., "The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications," *Proc. of the 16th ACM Symposium on Operating System Principles*, October 1997.

[Nob97] Noble, B.D., Satyanarayanan, M., Narayanan, D., Tilton, J.E., Flinn, J., Walker, K.R., "Agile Application-Aware Adaptation for Mobility," *Proc. of the 16th ACM Symposium on Operating System Principles*, October 1997.

[Ols99] Olson, M., Bostic, K., Seltzer, M., "Berkeley DB," *Proc. of the USENIX 1999 Annual Technical Conference*, June 1999.

[Par93] Parekh, A., Gallagher, R., "A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks—the Single Node Case," *ACM/IEEE Transactions on Networking*, 1(3):344-357, June 1993.

[Pet99] Petrou, D., Milford, J., Gibson, G., "Implementing Lottery Scheduling: Matching the Specializations in Tradi-

tional Schedulers," *Proc. of the USENIX 1999 Annual Technical Conference*, June 1999.

[Sel96] Seltzer, M., Endo, Y., Small, C., Smith, K., "Dealing with Disaster: Surviving Misbehaved Kernel Extensions," *Proc. of the Second Symposium on Operating System Design and Implementation*, October 1996.

[Sma98] Small, C., *Building an Extensible Operating System*, Ph.D. thesis, Division of Engineering and Applied Sciences, Harvard University, October 1998.

[Ste99] Steere, D.C., Goel, A., Gruenberg, J., McNamee, D., Pu, C., Walpole, J., "A Feedback-Driven Proportion Allocator for Real-Rate Scheduling," *Proc. of the Third Symposium on Operating Systems Design and Implementation*, February 1999.

[Sto96] Stoica, I., Abdel-Wahab, H., Jeffay, K., Baruah, S., Gehrke, J., Plaxton, C.G., "A Proportional-Share Resource Allocation Algorithm for Real-Time, Time-Shared Systems," *Proc. of the IEEE Real-Time Systems Symposium*, December 1996.

[Sul99] Sullivan, D., Haas, R., Seltzer. M., "Tickets and Currencies Revisited: Extending Multi-Resource Lottery Scheduling," *Proc. of the Seventh Workshop on Hot Topics in Operating Systems*, March 1999.

[Sun98] "Solaris Resource Manager 1.0: Controlling System Resources Effectively: A White Paper," http://www.sun.com/software/white-papers/wp-srm/.

[Ver98] Verghese, B., Gupta, A., Rosenblum, M., "Performance Isolation: Sharing and Isolation in Shared Memory Multiprocessors," *Proc. of the Eighth Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[Wal94] Waldspurger, C.A., Weihl, W., "Lottery Scheduling: Flexible Proportional-Share Resource Management," *Proc. of the First Symposium on Operating System Design and Implementation*, November 1994.

[Wal95] Waldspurger, C.A., "Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management," Ph.D. thesis, MIT/LCS/TR-667, MIT Laboratory for Computer Science, September 1995.

[Wal96] Waldspurger, C.A., Weihl, W., "An Object-Oriented Framework for Modular Resource Management," *Proc. of the Fifth Int'l Workshop on Object Orientation in Operating Systems*, October 1996.

[Wei84] Weicker, R.P., "Dhrystone: A Synthetic Systems Programming Benchmark," *Communications of the ACM*, October 1984.